345 Ludic Computing

Lecture 10

Pathfinding

Simon Colton & Alison Pease
Computational Creativity Group
Department of Computing
Imperial College London

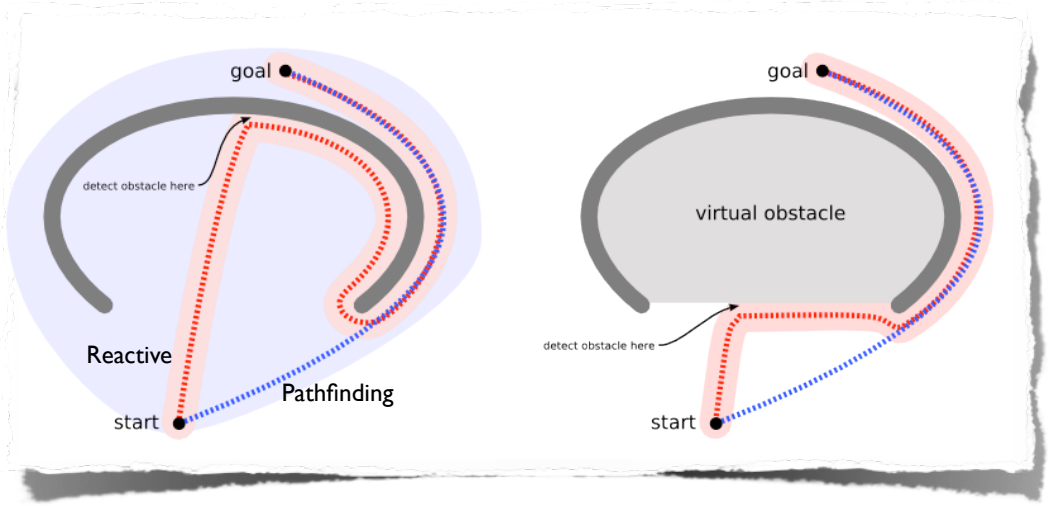
www.doc.ic.ac.uk/ccg

Pathfinding

Planned Movement

- Agent reasons about a complex representation of the world in order to construct and execute a navigation plan
- **Intelligent** and **reliable**: can be designed to provide solutions that are optimal and/or within given constraints
- **No ideal approach**: each game will require a particular range of functionality within limited resources
- **Hybrid approaches** are popular: pathfinding for the big picture, plus reactive local movement that can respond quickly to change

The Need for Pathfinding



- Reactive agents don't plan movement and are easily trapped. Annotating the environment can help, e.g. identify and avoid convex objects
- But in complex game environments agents often need to plan ahead

Image source: Patel

This Lecture

- Division schemes — Representation
  - Tiles, waypoints, navmeshes
- Pathfinding — Search
  - A* / Hierarchical A*
  - Precomputing paths

Navigation Graphs

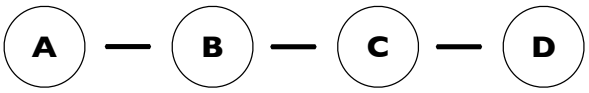
- Pathfinding relies on **search** over a **navigation graph**: an abstract (and hence imperfect) **representation** of the game world as a set of nodes and weighted edges
- Crafting a good representation (graph) is crucial for search performance and path quality, and can be very labour-intensive
    - Navigation is now largely a *content creation* problem
    - Developers use a mixture of manual and automatic techniques to create high quality navigation graphs

Division Schemes

- A **division scheme** is required to map the game world into a *set of linked regions* that define a *navigation graph*
- Game worlds often represented as "polygon soup"
    - Doesn't encode navigable areas vs. obstacles
    - Contain tiny details irrelevant to pathfinding
    - A division scheme gives a more informative representation for navigating the world

Game World

Division Scheme



Navigation Graph

Division Schemes

Quantisation & Localisation

- *Quantisation*: point in world ⇒ navgraph node
  - For an NPC to turn on a light switch the pathfinder has to *quantise* the NPC's and light switch's locations, and plan a path between those nodes
- *Localisation*: navgraph node ⇒ point in world
  - Given a node path the NPC has to *localise* each node and move between the resulting locations

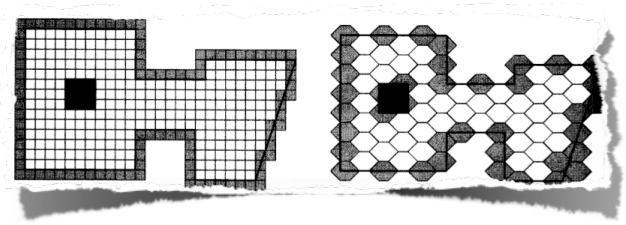
Division Schemes

Correctness & Validity

- *Validity*: if two points are in linked regions then there is a navigable line between them (the graph doesn't mislead)
  - *Invalid schemes result in paths that agents can't follow*
- *Completeness*: if there is a navigable line between two points then the line corresponds to a sequence of linked regions (the graph doesn't omit)
  - *Incomplete schemes prevent agents taking valid paths*
- Neither is enforced in practice, hence developers need to debug navigation graphs

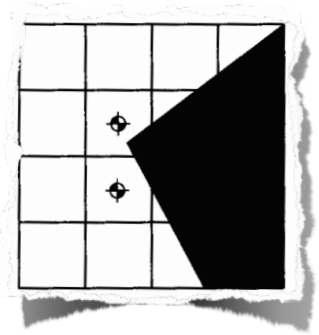
Tiles

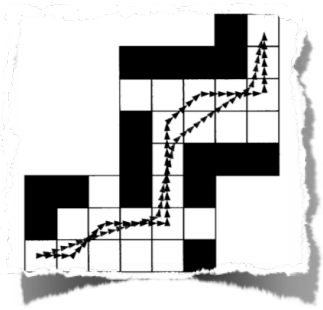
- Tile grids still underlie many games (e.g. some RTSs)
- World is divided into regular regions (triangle, hexagon or square) forming a regular navigation graph (degree 3, 6 or 8)
- Constant time quantisation (due to regularity) and localisation (pick centre of tile)
- Navigation graphs can be generated automatically on-the-fly



Tiles
Drawbacks

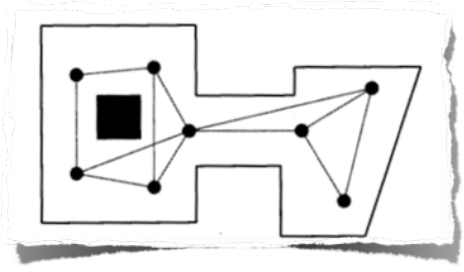
- Huge numbers of tiles can be generated
  - Each is a node that slows down search, but may not represent an interesting alternative path, e.g. slightly different paths through a large open area
- If tiles can be partially blocked then the graph can be invalid
- Paths often appear very irregular and unrealistic (though can be smoothed)

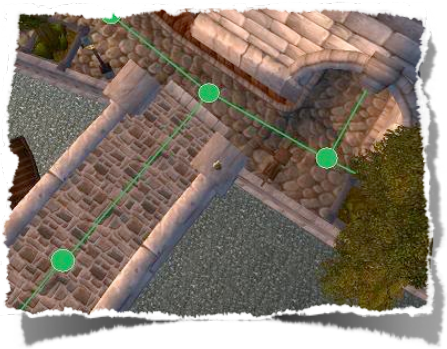




Waypoints

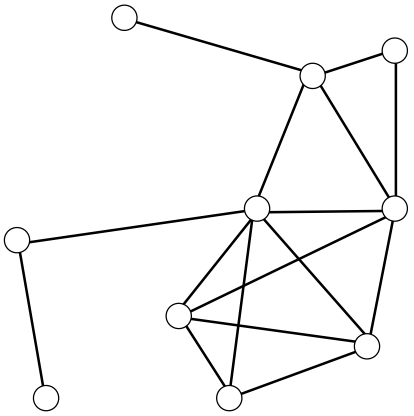
- Populate world with **waypoints** and navigable connections between them
- Embeds a navigation graph into the world
- Localisation is trivial (nodes are waypoints)
- Quantisation requires a search for the nearest waypoint, but we need to **validate** movement to the waypoint





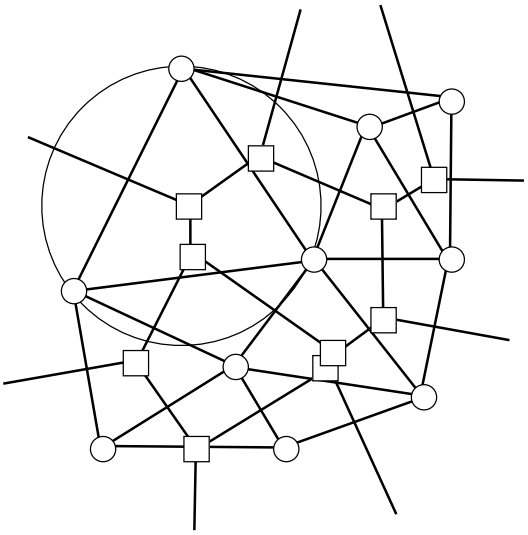
Waypoints
Construction

- Level designers often manually place waypoints (yellow circles)
- Then manually place connections. Danger of creating invalid navgraph. Check each connection!
- Can automate connections: check every pair of waypoints for a valid path
- But this produces too many edges and slows search



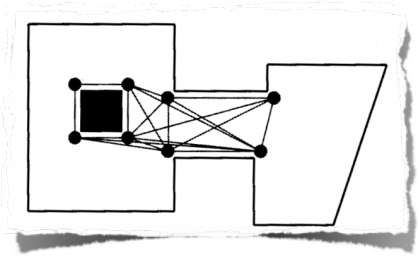
Waypoints
Triangulation & Quantisation

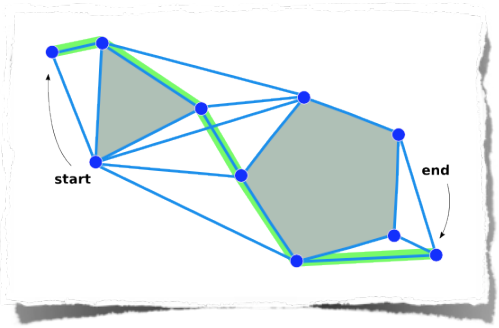
- A better connection method is the *Delaunay triangulation*
  - Draw circumcircle for any three points
  - If no waypoint lies within circle then connect this triangle (check each connection is valid)
- Centres of these circumcircles (red squares) form the *dual graph* of the triangulation
  - Known as the *Voronoi tessilation*: defines the quantisation regions for each waypoint!



Waypoints
Points of Visibility

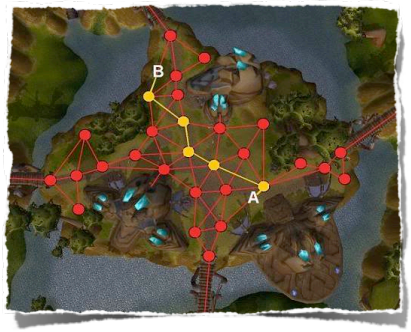
- An automatic method for waypoint generation that creates **corner graphs**
- A waypoint is placed at every convex corner and connections are added (either pair-wise or via triangulation)
- Move points inwards to avoid wall hugging, but paths can still be unrealistic

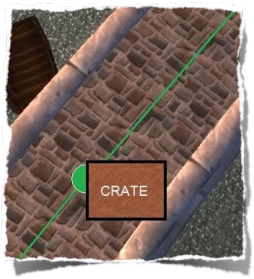




Waypoints
Drawbacks

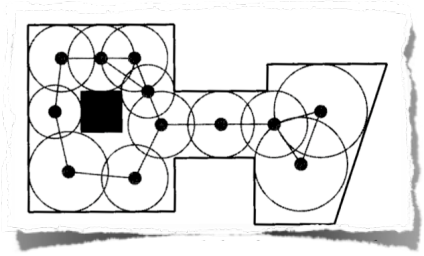
- Some areas require a huge number of waypoints and connections to generate quality paths, which slows down search
- Paths can be irregular, and smoothing requires **validation** (as did quantisation and triangulation!)
  - Without validation the agent could fall off a cliff or hit a wall
- Blocking a waypoint or connection can prevent navigation, even if the agent could get past the obstacle
- Separate graphs are required for different sized agents

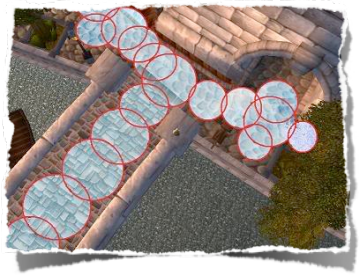




Waypoints
Annotations

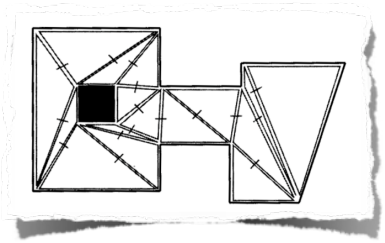
- The fundamental problem is that the waypoint graph doesn't represent anything off the network
- We can annotate waypoints with circles that represent a navigable area around waypoint. Connections are formed by overlapping circles
- Circles are bad at covering angular worlds. More circles give better representation, but at the cost of slower search



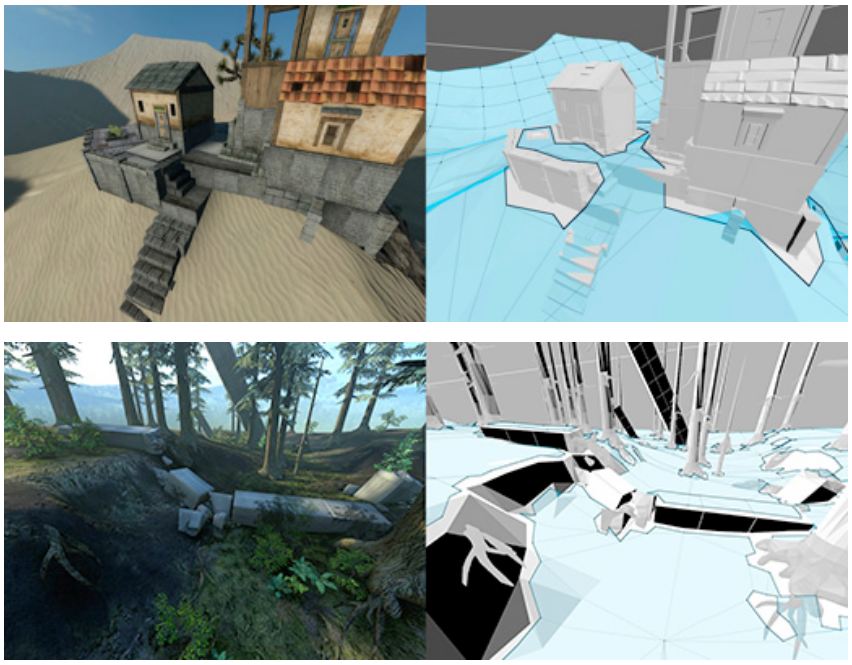


Navigation Meshes

- A set of convex polygons (often triangles) that describe the **navigable surfaces** of the world
- Convexity allows direct movement between points within a polygon
- Combine the flexibility of waypoint graphs with coverage (and hence validity) of tiles
- Quality paths can be found with smaller navigation graphs, leading to faster search



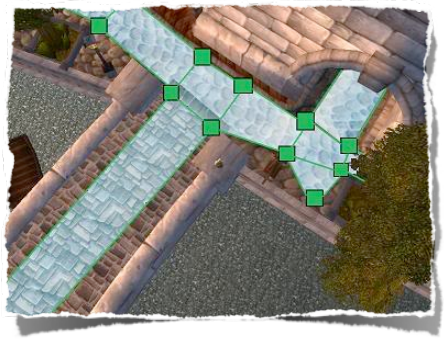
Navigation Meshes

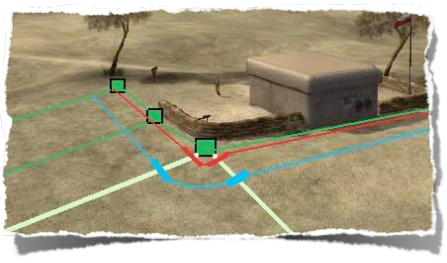


Lugaru http://blog.wolfire.com/

Navigation Meshes

- The localisation of a polygon is its geometric centre (average of the vertices)
- Quantising a point requires a search for the polygon that contains it
- Agents of different sizes can use same mesh by ensuring they stay a safe distance from the edge of the mesh
- Meshes can be constructed automatically from level geometry

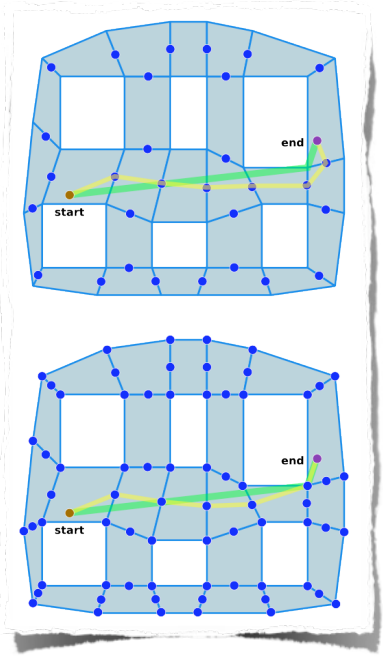




Navigation Meshes

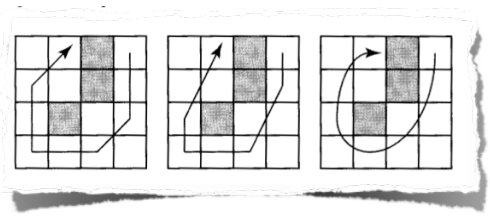
Pathfinding

- Pathfinding returns a series of polygons rather than a path of points an agent can use
- Polygons can be crossed using edge midpoints. Adding vertex points improves cornering
- Path smoothing (green) can improve paths. The path remains valid as long as the same polygons are used



Path Smoothing

- All division schemes can give erratic, unrealistic looking paths
- **String pulling** checks three consecutive points at a time: if the line from the first to the third is navigable, then the second is removed
- A smooth path that passes through the remaining points can be computed, e.g. using Catmull-Rom splines



Left: original path
Middle: after string pulling
Right: a Catmull-Rom spline

A* Pathfinding

Basics

- Navigation graph, start node A, goal node B
- For a path $P = N_1...N_k$ of nodes N_i
  - *Path cost* g(P) = sum of the weights for all path edges N_iN_{i+1} (i = 1 to k-1)
  - *Estimated remaining cost* h(P) = heuristic estimate of going from N_k to B
  - *Estimated cost* f(P) = g(P) + h(P)
- Example heuristic: the straight line distance to goal

A* Pathfinding

Data Structures

- A* pathfinding on a arbitrary graph requires the following data structures
  - `Paths`: a priority queue of paths ordered (cheapest first) by estimated cost f(P)
  - `Explored`: a set of explored nodes
  - `Least`(P): the lowest estimated cost f(P) found so far for each P in `Explored`

A* Pathfinding
Search Algorithm

`Paths` = {A}, `Explored` = {A} and `Least`(A) = f(A)

While `Paths` is non-empty

- Get the next path P = $N_1...N_k$ from `Paths`
- $M_1,...,M_q$ are the neighbours of N_k. For each M_i:
  - If M_i = B then *terminate with solution* $N_1...N_kB$
  - Else if $M_i \in$ `Explored` then *update* `Least`(M_i) if necessary
  - Else add M_i to `Explored` and *insert* $N_1...N_kM_i$ into `Paths`

A* Pathfinding
Complete and Optimal

- **Complete**: will find a path if one exists
- Finds **optimal** (lowest cost) paths if h(X) is an *admissible heuristic*: it always underestimates the true cost from X to B
  - Admissible heuristics are always optimistic
  - If a pathfinding heuristic is only slightly over-estimating it may still perform well and produce near-optimal paths

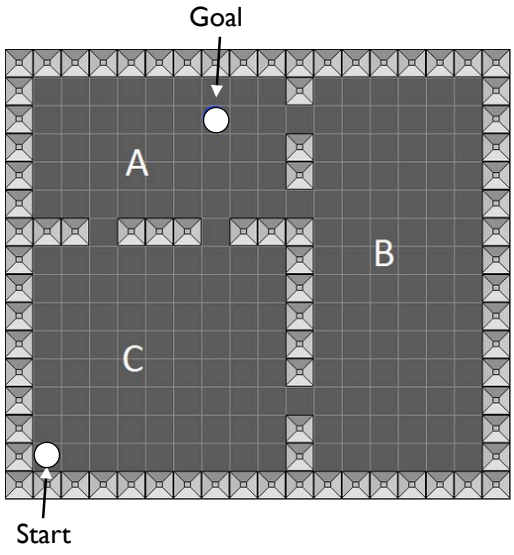
Hierarchical Pathfinding

- Performance of A* decreases with size of navigation graph. Large graphs can result in performance bottlenecks
- Hierarchical search uses an abstract graph to reduce search
- Intuition: If you want to travel from New York to California, you first plan the states you want to travel through, then the actual roads
- The hierarchy can be more than 2 levels deep

HPA* Example

Original Problem

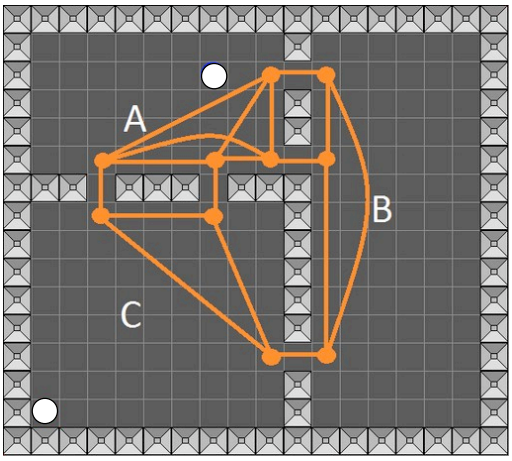
- HPA* is a hierarchical pathfinding algorithm for any navigation graph
- Example results: 1% near optimal paths, x10 less effort
- This example illustrates its use on a square grid. Circles mark Start and Goal nodes



HPA* Example

The Abstract Graph

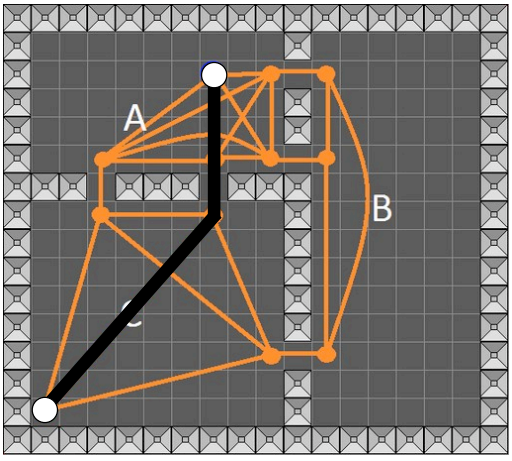
- A* must search from tile to tile
- HPA* is using a pre-computed abstract graph (a 2 level hierarchy)
- Low-level nodes are grouped into clusters A, B and C
- Each cluster is represented by a subset of its nodes that border other clusters. Each subset forms a *clique* in the abstract graph



HPA* Example

Search

- The start and goal nodes are inserted into the abstract graph, using A* to connect the new node to all members of its clique
- A* is applied to the modified abstract graph, giving an abstract path (thick line)
- The abstract path is refined into a path on the original graph, using either A* or cached paths between abstract nodes
- The path is smoothed to remove irregularities



Variations on A*

- Countless variations on pathfinding A* and heuristics, each tailored to specific contexts
- Single start, multiple goal queries (open goal A*)
- Dynamic navigation graphs, e.g. D*
- Low memory for mobile gaming, e.g. IDA*, SMA*
- Interruptible for limited resources per frame, e.g. LPA*, SHPA*

Pathfinding Queries

- A pathfinder may have to solve several types of **query**, depending of the requirements of the game
- **Locations**: the path start and goal may be single points or point sets (point-to-point, point-to-set, set-to-point, set-to-set)
- **Agents**: may need to plan a single agent, one-of-many, many, or groups. Agents may have different movement abilities, e.g. a tank can't get through narrow gap
- **Cost**: least cost paths (shortest, least effort, safest), maximum reward per cost paths
- **Urgency**: quick poor path now vs. quality path later

Pathfinding Under Pressure

- With large numbers of queries and limited CPU time, pathfinders often don't have time to search for a detailed path before a solution is needed
- Instead, it can find a **quick path** with (say) 10 iterations of A*, just to get the agent moving
- Now there is time to plan a **full path** to the goal
- A **splice path** is required to connect the already executed segment of the quick path to the full path
- Pathfinding jobs can be prioritised, e.g. if an agent is near the end of its quick path and no full path is ready

Pre-computing Paths

To Search Or Not To Search?

- Pre-computed optimal paths can be stored for some or all of a navigation graph
    - Benefit: little or search required during play
    - Costs: **memory** + precomputation
- Store as 'next node' look-up table
    - Given current and goal node, returns next node in path
    - Finding an N node path requires only N look ups
    - Table requires $O(nodes^2)$ memory

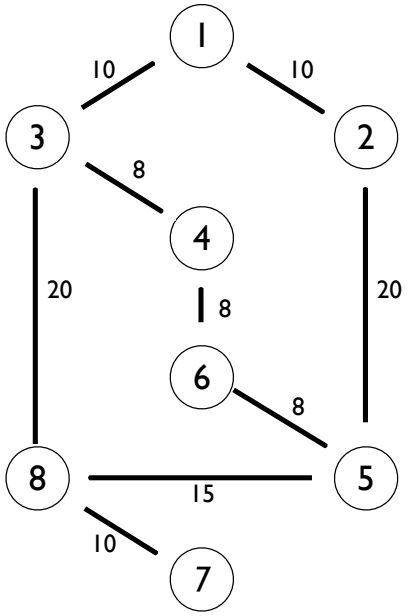
Pre-computing Paths

A Next Node Table

Goal node

Current node

	1	2	3	4	5	6	7	8
1	-	2	3	3	2	3	2	2
2	1	-	1	1	5	1	5	5
3	1	1	-	4	4	4	4	4
4	3	3	3	-	6	6	6	6
5	2	2	8	8	-	8	8	8
6	4	5	4	4	5	-	5	5
7	8	8	8	8	8	8	-	8
8	3	5	3	3	5	3	7	-



Pre-computing Paths

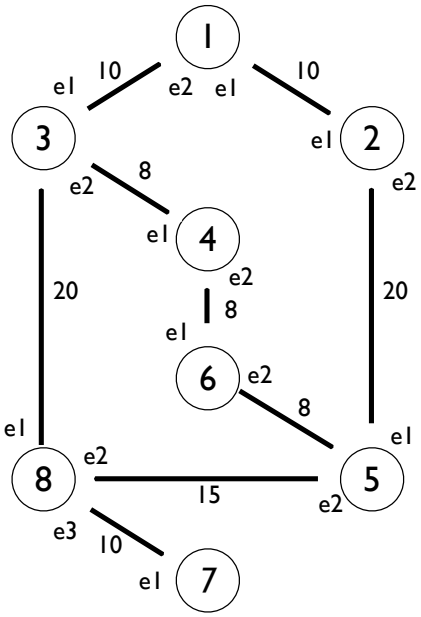
A Next Edge Table

Goal node

Current node

	1	2	3	4	5	6	7	8
1	-	e1	e2	e2	e1	e2	e1	e1
2	e1	-	e1	e1	e2	e1	e2	e2
3	e1	e1	-	e2	e2	e2	e2	e2
4	e1	e1	e1	-	e2	e2	e2	e2
5	e1	e1	e2	e2	-	e2	e2	e2
6	e1	e2	e1	e1	e2	-	e2	e2
7	e1	e1	e1	e1	e1	e1	-	e1
8	e1	e2	e1	e1	e2	e1	e3	-

Storing edge indices ("take the 3rd edge") rather than node labels requires less memory, as the maximum degree is typically much less than the number of nodes



Summary

- Division scheme is crucial to success
  - Tiles, waypoints, navigation meshes
- Search typically based on A*
  - Depends on a heuristic
  - Hierarchical search with HPA*
  - Off-line pathfinding

Reading and Resources
(Optional)

- Amit Patel's A* pages http://theory.stanford.edu/~amitp/GameProgramming/
- Recast & Detour http://code.google.com/p/recastnavigation/
- Mat Buckland (2005) *Programming Game AI by Example*
- Ian Millington & John Funge (2009) *AI for Games*, 2nd edition
- AIGameDev.com